

IKEv2 Mobility and Multihoming
(mobike)
Internet-Draft
Expires: January 19, 2006

U. Schilcher
H. Tschofenig
F. Muenz
Siemens AG
July 18, 2005

Application Programming Interface to a Trigger Database for MOBIKE
draft-schilcher-mobike-trigger-api-01.txt

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with [Section 6 of BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on January 19, 2006.

Copyright Notice

Copyright (C) The Internet Society (2005).

Abstract

The purpose of MOBIKE is the creation and maintenance a set of available addresses and provide them to the communication partner. A MOBIKE peer should have some information about the status of each address and interface in order to execute the respective actions. Examples may comprise switching from address or interface to another. This information, which will be referred as trigger, is distributed over a number of protocols daemons at an end host. To make this

information available to the MOBIKE daemon it is necessary to store it centrally at the host (called trigger database) and to enable the protocols to insert the triggers and to allow MOBIKE to obtain timely information.

Table of Contents

1.	Introduction	3
2.	Terminology	4
3.	Trigger Classification	5
4.	API for the Trigger Database	6
5.	Supported Message Types	7
6.	Payload Format	12
7.	Applicability	17
8.	IANA Considerations	18
9.	Security Considerations	19
10.	Acknowledgments	20
11.	References	21
11.1	Normative References	21
11.2	Informative References	21
	Authors' Addresses	21
	Intellectual Property and Copyright Statements	23

[1.](#) Introduction

When a MOBIKE implementation is started first it has to build a set of all available addresses (or a subset of them for policy reasons; see [\[3\]](#)) before communicating with another peer. From these addresses, it has to select one of the addresses as preferred address that will be used as the source address in the communication with the MOBIKE peer.

This address set together with the preferred address may change during operation because of several reasons, e.g. an interface could be disconnected or the communication path becomes unavailable due to router failure. Many of the events, which cause the change of the address set, are out of the scope of the MOBIKE protocol itself but need an interaction with other protocols daemons locally at the end host.

For MOBIKE to work, it is really important to know about the status of the available addresses in order to make reasonable decisions. A number of other protocols running on the end host might have various information necessary to derive a decision whether to switch from one preferred address to another or whether it is necessary to modify the peer address set.

In this document, we therefore suggest to define an API that allows protocol daemons to insert information (triggers) into a "database" that can later be made available to the MOBIKE daemon. The API is based on the BSD routing socket API in a similar fashion as PF_KEY [\[1\]](#) extends the same API for generic key management usage. This document therefore heavily focuses on the functionality offered by the PF_KEY specification.

[2.](#) Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [\[2\]](#).

Additionally, the following terms are introduced:

- o Trigger: Information which is relevant for MOBIKE about an address.
- o Trigger Database (TDB): Collection of triggers which can be accessed via the API defined in this document.

[3.](#) Trigger Classification

Many different events may cause a change in the address set used by MOBIKE (see [\[3\]](#)). These events can be notified by many different protocols running in kernel or user space. Since the reaction (if any) on a given event depends on the type of the event, a classification of these events is necessary.

As an example, we define the following triggers in this document:

Trigger type	Value	Description
TDB_TTYPE_IF_ADDED	1	New interface added
TDB_TTYPE_IF_REMOVED	2	Interface removed
TDB_TTYPE_IF_ADDRADDED	3	New address added to interface
TDB_TTYPE_IF_ADDRREMOVED	4	Address removed from interface
TDB_TTYPE_IF_ADDRCHANGED	5	Interface has changed one of its addresses (e.g. new DHCP lease)
TDB_TTYPE_TUNNEL_ADDED	6	IPSec tunnel was established
TDB_TTYPE_TUNNEL_CHANGED	7	IPSec tunnel conf. changed
TDB_TTYPE_TUNNEL_REMOVED	8	IPSec tunnel was removed
TDB_TTYPE_CONN_ESTABLISHED	9	e.g. dial-in network

			has connected
TDB_TTYPE_CONN_LOST		10	connection to network lost
TDB_TTYPE_DEST_UNREACHABLE		11	e.g. ICMP packet received
TDB_TTYPE_MAX		12	Maximum value for trigger types

A future version of this document will add more triggers and a more detailed description of them. The types TDB_TTYPE_TUNNEL_ADDED, TDB_TTYPE_TUNNEL_CHANGED and TDB_TTYPE_TUNNEL_REMOVED are inspired by [4].

The above listed trigger types will be signaled using the "tdb_trigger" message structure described in [Section 6](#)

[4.](#) API for the Trigger Database

To access the trigger database, an API is defined. For that purpose the new network protocol family ID PF_TRIGGER has to be defined. The operation of the API is analogue to the PF_KEY interface (see [1]).

To access the API, a socket of the family PF_TRIGGER has to be created. To communicate with the Trigger Database, messages are sent and received through the socket with the send and recv commands. Any other commands like bind, connect, etc. are not supported and MUST NOT have any effects on a socket of the PF_TRIGGER family.

The following exhibits an example socket creation:

```
int s = socket(PF_TRIGGER, SOCK_RAW, PF_TRIGGER);
```

The format of the messages is the following: Each message starts with a fixed header. Appended to this header, there are some payloads depending on the type of the message. The available message types are described in [Section 5](#).

Each time when a message is sent to the Trigger Database, it will respond with a message of the same type. This response contains the same payloads as transmitted to the Trigger Database, only some additional information MAY be included (e.g., the Trigger Database assigns an id to each trigger).

The normal operation works in the following way: A MOBIKE implementation, which wants to be informed about every new trigger, registers itself to the Trigger Database by sending a TDB_REGISTER message. If a protocol daemon wants to add a new trigger, it sends a TDB_ADD message to the Trigger Database including information that is important for this new trigger.

The Trigger Database acknowledges this message with a TDB_ADD response to the network protocol and with a TDB_NOTIFY message to the registered MOBIKE implementation. This notify message contains some information about the new trigger including its id. All information available about the new trigger can be requested with a TDB_GET message.

In a future version of this document, we will try to add some information about scenarios to better illustrate the interaction.

[5](#). Supported Message Types

Several different message types can be sent to the Trigger Database using a PF_TRIGGER socket. The message type is indicated by the tdb_header_msgtype field that is part of the generic message header (see [Section 6](#)) and can be one of the following values:

Message type	Value	Description
--------------	-------	-------------

TDB_ADD	1	Add a trigger to the Trigger Database
TDB_GET	2	Get information about an existing trigger.
TDB_DELETE	3	Delete a trigger from the Trigger Database
TDB_REGISTER	4	Register an application to receive a messages for each new trigger added.
TDB_NOTIFY	5	A new trigger has been added, deleted or updated.
TDB_MODIFY	6	Modify a trigger in the Trigger Database
TDB_DUMP	7	Dump all Trigger Database entries
TDB_FLUSH	8	Delete all Trigger Database entries
TDB_MAX	9	Generic maximum for message types

Each message type requires different payloads to be appended. Each payload starts with a generic payload header followed by payload specific data. The generic header has the following structure:

```
struct tdb_payload {
    uint16_t      tdb_payload_len;
    uint16_t      tdb_payload_type;
} __attribute__(( packed ));
/* sizeof( struct tdb_payload ) == 4 */
```

The tdb_payload_len field contains the length of the payload divided by 8. The type of the payload is determined by the tdb_payload_type field, which contains one of the following values:

Payload type	Value	Description
TDB_PT_INTERFACE	1	Information about an interface
TDB_PT_ADDRESS	2	The IP address of an IF
TDB_PT_TRIGGER	3	Trigger id, type, etc.

found below:

TDB_ADD:

If an application or network protocol wants to add a new trigger, it sends a TDB_ADD message to the Trigger Database. The new trigger is stored in the Trigger Database and a corresponding TDB_NOTIFY message that indicates that a new trigger has been added is sent to all registered applications.

The format of the message is:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

The TRIGGER payload indicates the type of the trigger and also includes some trigger specific data. The INTERFACE payload is needed to select the appropriate hardware interface, the new trigger is related to. For many triggers, an additional address payload is required. It contains, for example, the new address for a TDB_TTYPE_IF_ADDRCHANGED trigger.

The response from the Trigger Database contains the same information as the request:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

TDB_DELETE:

A trigger, which is stored inside the Trigger Database, can be deleted using the TDB_DELETE payload. In the request the only information, which has to be specified is the id of the trigger, which is stored in 'TRIGGER(*)'.

The format of the message is:

<HEADER, TRIGGER(*)>

The Trigger Database responds with a message with the following format:

<HEADER, TRIGGER>

In the response, the TRIGGER payload has all fields filled with the correct values.

TDB_GET:

The TDB_GET message is used to request all available information of a specified trigger. In the request the only information, which has to be specified is the id of the trigger, which is stored in 'TRIGGER(*)'.

The format of the message is:

<HEADER, TRIGGER(*)>

The Trigger Database responds with a message of the following format:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

In the response a fully initialized TRIGGER payload is present. Additionally, INTERFACE payload is present as well as and an optional an ADDRESS payload, if an address is available for the specified trigger.

TDB_REGISTER:

An application, which is interested in each new trigger, can register itself to the Trigger Database. After the application has registered, it receives a message each time a new trigger has been added to the database. The format of the message is:

<HEADER>

No additional payload has to be added. The Trigger Database responds with a message of the same type and with the same content, i.e. its format is:

<HEADER>

TDB_NOTIFY

An application that has registered itself to get informed about the new triggers or updates to these triggers, receives a TDB_NOTIFY message. The format of the message is the same as for a TDB_ADD message. The only difference is that some field are filled by the Trigger Database before sending the TDB_NOTIFY message.

The format of the message is:

<HEADER, TRIGGER, [INTERFACE], [ADDRESS]>

Since this message is sent by the Trigger Database itself, a registered application MUST NOT respond to it.

TDB_MODIFY:

If an application or a network protocol wants to modify a new trigger (because its status has changed), it sends a TDB_MODIFY message to the Trigger Database. The new trigger is stored and a corresponding TDB_NOTIFY message that indicates that an existing trigger has been modified is sent to all registered applications.

The format of the message is:

<HEADER, TRIGGER, [INTERFACE], [ADDRESS]>

The TRIGGER payload indicates the type of the trigger and also includes some trigger specific data.

The response from the Trigger Database contains the same information as the request:

<HEADER, TRIGGER, [INTERFACE], [ADDRESS]>

TDB_DUMP:

An application, that wants to learn all currently available triggers should send a TDB_DUMP message. Since a TDB_GET message requires a specific trigger id for retrieval, applications which do not know all trigger ids depend on this message class for learning all unknown triggers. The format of the message is:

<HEADER>

The Trigger Database will respond with all currently available triggers entries by serially sending the following message:

<HEADER, TRIGGER, INTERFACE, [ADDRESS]>

TDB_FLUSH:

For deleting all entries in a Trigger Database, the TDB_FLUSH message is used. Since the TDB_GET message requires a specific trigger id for deletion, reliable cleaning of a Trigger Database can be done with this message. The format of the message is:

Schilcher, et al.

Expires January 19, 2006

[Page 10]

Internet-Draft

API to a Trigger Database for MOBIKE

July 2005

<HEADER>

The Trigger Database will respond with the following message:

<HEADER>

[6.](#) Payload Format

HEADER:

Each message starts with the fixed header. It contains general information about the message and determines, which payloads have to be included in it. It has the following format:

```
struct tdb_header {
    uint8_t      tdb_header_version;
    uint8_t      tdb_header_msgtype;
    uint8_t      tdb_header_errno;
    uint8_t      tdb_header_reserved1;
    uint16_t     tdb_header_msglen;
    uint16_t     tdb_header_reserved2;
    uint32_t     tdb_header_seq;
    uint32_t     tdb_header_pid;
} __attribute__(( packed ));
/* sizeof( struct tdb_header ) == 16 */
```

The fields of this structure contain the following values:

`tdb_header_version`: The version of the used PF_TRIGGER interface.

This document specifies this API in version 1.

`tdb_header_msgtype`: This field contains the type of the message.
All possible values are listed in the table in [Section 5](#).

`tdb_header_errno`: If an error occurred while processing a request, the response will only include the message header without any payloads. The type of the error is indicated by the value in this field. The values are taken from the error number specification of the operating system (e.g. the `errno.h` file).

`tdb_header_msglen`: The length of the message divided by 8 is stored into this field.

`tdb_header_seq`: This field contains the number of the last message sent incremented by 1.

`tdb_header_pid`: The process id of the program sending the message.
If the message is generated inside the kernel, this value is set to zero.

INTERFACE:

The INTERFACE payload is used to provide all needed information about an active network interface.

The format of the INTERFACE payload is the following:

```
struct tdb_interface {
    uint16_t      tdb_interface_len;
    uint16_t      tdb_interface_pltype;
    uint32_t      tdb_interface_selector;
    uint32_t      tdb_interface_type;
    uint32_t      tdb_interface_quality;
```

```
} __attribute__((packed));  
/* sizeof( struct tdb_interface ) == 16 */
```

This fields contain the following values:

`tdb_interface_len`: This field contains the length of the payload divided by 8.

`tdb_interface_pltype`: This field contains the value `TDB_PT_INTERFACE`.

`tdb_interface_selector`: The `tdb_interface_selector` field stores interface enumeration information for unique identification (IF #0, #1, #2, ...). When a new interface comes up, this value should be set by the kernel.

`tdb_interface_type`: Information about classification of an interface, for instance Indication, of fixed or wireless network link and theoretical maximum bandwidth.

`tdb_interface_quality`: This field provides quality information about a certain interface for making interface selections possible (e.g. load balancing; handover). This value should be a very general indicator calculated and set by the kernel

space. It could be based on latency (ping), signal quality for wireless links, packet-loss rate and average data-throughput/bandwidth. (Author's note: If a single value is not reasonable, separate indicators for all these evaluation criteria's should be defined.)

Further information about an interface might be necessary. This is left for future investigation.

ADDRESS:

The ADDRESS payload is used to provide the IP address of an interface to the Trigger Database or registered application. This information is important for most triggers. But it might be possible that there trigger types that do not need an ADDRESS payload.

The format of the ADDRESS payload is:

```
struct tdb_address {
    uint16_t tdb_address_len;
    uint16_t tdb_address_pltype;
    uint8_t tdb_address_proto;
    uint8_t tdb_address_prefixlen;
    uint16_t tdb_address_reserved;
} __attribute__(( packed ));
/* sizeof( struct tdb_address ) == 8 */

/* followed by some form of struct sockaddr */
```

Information about IP address and probably ports is provided by a sockaddr structure which is attached to the tdb_address structure. A sockaddr structure is capable of storing both a IPv4 and IPv6 address. The fields of the tdb_address structure contains the following values:

tdb_address_len: This field contains the length of the payload including the sockaddr structure divided by 8.

tdb_address_pltype: The tdb_address_pltype field contains the value TDB_PT_ADDRESS.

tdb_address_proto: The tdb_address_proto field is normally set to zero. However, if is are set in the attached sockaddr needed, then the field SHOULD be set to the protocol number of the upper layer protocol. (e.g. TCP or UDP). This functionality

may become relevant for signaling IPsec related information (e.g. tunnel changes)

`tdb_address_prefixlen`: This field contains the prefix length of the address.

`tdb_address_reserved`: The `tdb_address_reserved` field is reserved for future use and MUST be set to zero.

TBD: Clarification about the prefix len needs to be provided in a future document version.

TRIGGER:

The TRIGGER payload is used to provide all needed information about a trigger itself, e.g. the trigger type, an id, etc. The notation TRIGGER(*) indicates that only the id field is used to identify the trigger and all other fields SHOULD be set to zero.

The format of the TRIGGER payload is the following:

```
struct tdb_trigger {
    uint16_t      tdb_trigger_len;
    uint16_t      tdb_trigger_pltype;
    uint16_t      tdb_trigger_type;
    uint16_t      tdb_trigger_reserved1;
    uint32_t      tdb_trigger_id;
    uint32_t      tdb_trigger_reserved2;
} __attribute__(( packed ));
/* sizeof( struct tdb_trigger ) == 16 */
```

This fields contain the following values:

`tdb_address_len`: This field contains the length of the payload divided by 8.

tdb_address_pltype: This field contains the value TDB_PT_TRIGGER.

tdb_address_type: The type of the trigger is stored into this field. All possible values are listed in the table in section [Section 3](#).

tdb_address_id: The id of a trigger is assigned by the Trigger Database itself. In the message sent by userspace programs, which do not know this value (e.g. for TDB_ADD messages), this value MUST be set to zero.

Further information about a trigger might be necessary. This is left for future investigation.

[7.](#) Applicability

Even though this document is intended to give a solution for MOBIKE, the API is generic enough to make information available for other protocols as well.

The Next Step In Signaling (NSIS) protocol suite, for example, requires access to up-to-date information about IP addresses, interfaces and interactions with mobility protocols. In order to react on mobility events some sort of interaction between the kernel, various signaling protocols (including Mobile IP, IKE/IPsec, etc.) and the NSIS daemon is required (see [\[5\]](#)). Hence, an NSIS daemon supporting mobility could benefit from a generic interface to meet it's requirements for fast and accurate detection of mobility events, address and interface changes. GIMPS, for example, demands immediate reaction in case of a mobility event (e.g., handover). Monitoring procedures of mobility management protocols like Mobile IP are required to react to these mobility events in an appropriate way.

The trigger database and it's API could provide necessary information for detecting such a movement (new interface/IP address available, expiring Mobile IP timers).

[8.](#) IANA Considerations

This document defines an IANA registry for the protocol family PF_TRIGGER.

An IANA registry might be needed for the different trigger types (for which examples are provided in [Section 3](#)).

[9.](#) Security Considerations

This document describes an API which allows information about IP addresses to be obtained at a local host. A malicious application or protocol daemon could disseminate wrong information. This would make other protocols, such as MOBIKE, believe that the status of a particular address has changed. This will likely lead to unexpected protocol behavior, such as switching between addresses back-and-forth. Hence, a certain trust has to be placed into the applications and protocol daemons that are allowed to access the database to insert, modify or delete triggers. Access control mechanisms might enforce certain rights to use the API or parts of it.

[10](#). Acknowledgments

The authors would like to thank Murugaraj Shanmugam for his comments.

[11.](#) References

[11.1](#) Normative References

- [1] McDonald, D., Metz, C., and B. Phan, "PF_KEY Key Management API, Version 2", [RFC 2367](#), July 1998.
- [2] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", March 1997.

[11.2](#) Informative References

- [3] Kivinen, T. and H. Tschofenig, "Design of the MOBIKE protocol", [draft-ietf-mobike-design-02](#) (work in progress), February 2005.
- [4] Sugimoto, S. and F. Dupont, "PF_KEY Extension as an Interface between Mobile IPv6 and IPsec/IKE", [draft-sugimoto-mip6-pfkey-migrate-00](#) (work in progress), February 2005.
- [5] Lee, S., Jeong, S., Tschofenig, H., Fu, X., and J. Manner, "Applicability Statement of NSIS Protocols in Mobile Environments", [draft-ietf-nsis-applicability-mobility-signaling-01](#) (work in progress), February 2005.

Authors' Addresses

Udo Schilcher
Siemens
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: USchilcher@siemens.com

Hannes Tschofenig
Siemens
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: Hannes.Tschofenig@siemens.com

Franz Muenz
Siemens AG
Otto-Hahn-Ring 6
Munich, Bayern 81739
Germany

Email: Franz.Muenz@thirdwave.de

Intellectual Property Statement

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Disclaimer of Validity

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Copyright Statement

Copyright (C) The Internet Society (2005). This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.

